© 2014 IJEDR | Volume 2, Issue 2 | ISSN: 2321-9939

Interrupt Controller for Digital Design using AMBA Protocol

[1]Avinash Gowda M, [2]Satheesh Rao
[1]PG scholar, [2]Asst. Professor
[1]Departement of E&CE,
[1]NMAM Institute of Technology, Nitte, India
[1]avinashgouda@gmail.com, [2]write2sat@gmail.com.com

***Abstract*— The Interrupt Controller is designed to interface with the AMBA bus. It can make the system more efficient and more responsive to critical events. Interrupt controller is designed with the concept of priority for immediate selection of peripherals which requires attention or service. The interrupt is necessary for any embedded processor based SOC design that is running a Real Time Operating System. The Interrupt Controller is fully scalable to support 32 interrupt sources and provides a programmable interrupt register which can be used to generate an interrupt under software control. Daisy chain concept is used to cascade the Interrupt Controller to increase the number of inputs. Here AHB is optimized to interface with Interrupt Controller to initiate data transfer on the AHB.**

***Index Terms*—AMBA, AHB, VIC, SoC, IP**

I. Introduction To AMBA

The Advanced Microcontroller Bus Architecture (AMBA) is used as the on-chip bus in system-on-a-chip (SoC) designs. Since its inception, the scope of AMBA has gone far beyond microcontroller devices, and is now widely used on a range of ASIC and SoC parts including applications processors used in modern portable mobile devices like Smartphone's.

AMBA is a registered trademark of ARM Limited, [1] and is an open standard, on-chip interconnect specification for the connection and management of functional blocks in a System-on-Chip (SoC). It facilitates right-first-time development of multi-processor designs with large numbers of controllers and peripherals.

Interrupt controller are important due to the fact that Processors and peripherals usually communicate with each other with interrupt [2]. With the development of SoC technique, the communication between processors and peripherals becomes a problem as processors have limited interrupt ports which is less than the total interrupt signals of peripherals and other processors.

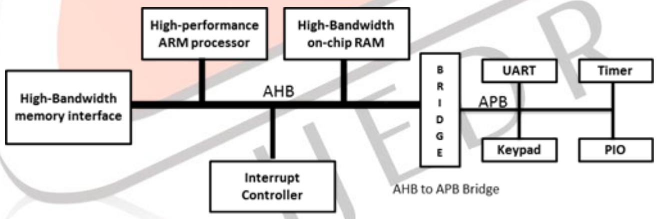

High-performance ARM processor
High-Bandwidth on-chip RAM
High-Bandwidth memory interface
AHB
Interrupt Controller
BRIDGE
AHB to APB Bridge
APB
UART
Timer
Keypad
PIO

Fig. 1. An Interrupt Controller in AMBA based system.

The important aspect of a SOC is not only which components or blocks it houses, but also how they interconnect. AMBA is a solution for the blocks to interface with each other.

An AMBA-based microcontroller typically consists of a high-performance system backbone bus (AMBA AHB), able to sustain the external memory bandwidth, on which the CPU, on-chip memory and other Interrupt Controller devices reside. This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers. Also located on the high performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located.

II. Methodology

In this project the RTL coding and formal verification is done using verilog HDL. Simulation is done using ISE Sim and corresponding synthesis is done with Xilinx ISE simulater.

III. Interrupt Controller Overview

Functional Overview

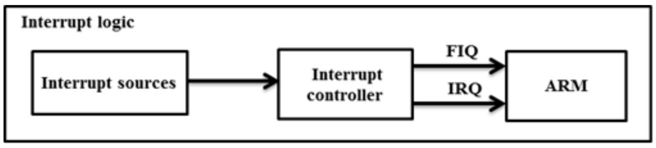


Fig. 2. Block diagram of Interrupt logic.

The Interrupt Controller is an Advanced Microcontroller Bus Architecture (AMBA) compliant System-on-Chip (SoC) peripheral. It is a programmable interrupt controller designed to interface to an AMBA AHB bus. The interrupt function is necessary for any embedded microprocessor based SOC that is running a Real Time Operating System (RTOS).

Interrupts are one of the most powerful and useful features one can employ in embedded systems. They can make the system more efficient and more responsive to critical events, and they can also make the software easier to write and understand. However, they can also be a confusing and error-prone feature in a program, and some people avoid them for that reason. But every embedded programmer should be at home with interrupts, should consider them as useful tools and not monsters in the closet, and use them whenever appropriate [1].

The Interrupt Controller is fully scalable to support 1 to 32 interrupt sources and provides a programmable interrupt register which can be used to generate an interrupt under software control.

Interrupt Controller

The Interrupt Controller is widely used for interfacing the processor with the peripheral devices. In SOC architecture, interrupts usually occurs from module sources, from external sources, and can also be generated by software with in the processor. The number of interrupts and other aspects can be tailored to the target system.

This Interrupt Controller handles multiple interrupt inputs from peripheral devices and provides a single interrupt output to the system processor and this is designed to interface with the AMBA protocol.

The Interrupt Controller provides an interface to the interrupt system and improves interrupt latency in two ways. They are,

- Moves the interrupt controller to the AMBA AHB bus.
- Provides vectored interrupt support for high-priority interrupt sources.

The Interrupt Controller provides a software interface to the interrupt system. In a system with an interrupt controller, software must determine the source that is requesting service and where its service routine is loaded. An Interrupt Controller does both of these in hardware. It supplies the starting address or vector address of the service routine, corresponding to the highest priority requesting interrupt source.

Interrupts In ARM

In an ARM system [3], two levels of interrupt are available. They are,

- Fast Interrupt request (FIQ) for fast and low latency interrupts handling.
- Interrupt Request (IRQ) for more general interrupts.

Generally, we only use a single FIQ source at a time in a system to provide a true low latency interrupt. This has the following benefits,

- We can execute the interrupt service routine directly without determining the source of the interrupt
- It reduces interrupt latency and we can use the banked registers available for FIQ interrupts more efficiently because we do not require a context save

The Interrupt Controller does not handle interrupt sources with transient behavior. For example, an interrupt is asserted and then disserted before software can clear the interrupt source. In this case, the CPU acknowledges the interrupt and obtains the vectored address for the interrupt from the AMBA-AHB Interrupt Controller, assuming that no other interrupt has occurred to overwrite the vectored address. However, when a transient interrupt occurs, the priority logic of the Interrupt Controller is not set and lower priority interrupts can interrupt the transient interrupt service routine, assuming interrupt nesting is permitted.

There are 32 interrupt lines. The Interrupt Controller uses a bit position for each different interrupt source. The software can control each request line to generate software interrupts. There are 16 vectored interrupts. These interrupts can only generate an IRQ interrupt. The vectored and non-vectored IRQ interrupts provide an address for an Interrupt Service Routine (ISR). Reading from the Vector Interrupt Address Register, VECT_addr, provides the address of the ISR, and updates the interrupt priority hardware that masks out the current and any lower priority interrupt requests. Writing to the VECT_addr Register indicates to the interrupt priority hardware that the current interrupt is serviced, enabling lower priority or the same priority interrupts to be removed, and for the interrupts to become active to go active [3].

IV. ARCHITECTURE OF INTERRUPT CONTROLLER

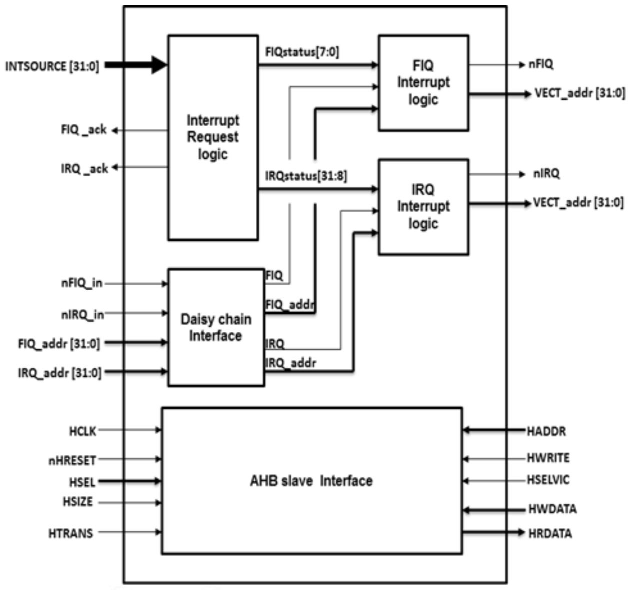


Fig. 3. Architecture of Interrupt Controller.

The Interrupt Controller is mainly divided into four blocks they are

- Peripheral interface.
- CPU interface.
- Daisy chain interface.
- AHB slave interface.

Peripheral Interface

1) Interrupt Request Logic

The interrupt request logic receives the interrupt requests from the peripheral and combines them with the software interrupt requests. It then masks out the interrupt requests that are not enabled, and routes the enabled interrupt requests to either IRQ_status or FIQ_status.

Fig.4. Shows a block diagram of the interrupt request logic.

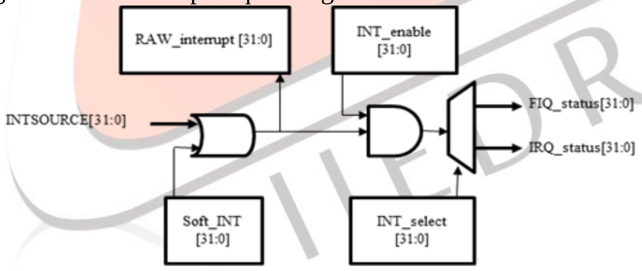


Fig. 4. Block of Interrupt Request Logic.

INTSOURCE [31:0] the interrupt request lines from the peripheral connect to the inputs of the Interrupt request logic.
RAW_interrupt [31:0] shows the status of the interrupts before masking by the enable registers. A HIGH bit indicates that the appropriate interrupt request is active before masking.
Soft_INT [31:0] setting a bit generates a software interrupt for the specific source interrupt before interrupt masking. A HIGH bit sets the corresponding bit in the Software Interrupt Register. A LOW bit has no effect.
INT_enable [31:0] enables the interrupt request lines.
On reset, all interrupts are disabled. A HIGH bit sets the corresponding bit in the Interrupt Enable Register. A LOW bit has no effect.
Vectored interrupts are only generated if the interrupt is enabled. The specific interrupt is enabled in the Interrupt Enable Register, and the interrupt is set to generate an IRQ interrupt in the Interrupt Select Register. This prevents multiple interrupts being generated from a single request if the controller is incorrectly programmed.

CPU Interface

1) FIQ Interrupt Logic

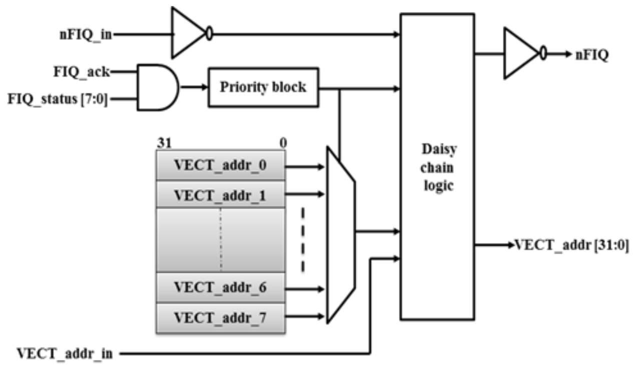


Fig. 5. Block of FIQ Interrupt Logic.

The FIQ interrupt logic generates the FIQ interrupt signal by combining the FIQ interrupt requests in the interrupt controller and any requests from daisy-chained interrupt controllers. Fig.5. shows a block diagram of the vectored FIQ interrupt logic.
Here we use fixed priority logic for upper 8 bits of INTSOURCE and generates the nFIQ signals which is active low and selects the VECT_addr of the respective peripheral from vectored table to the CPU.
FIQ_status acts as a select line for vectored address selection. nFIQ is active low signal for CPU.

2) IRQ Interrupt Logic

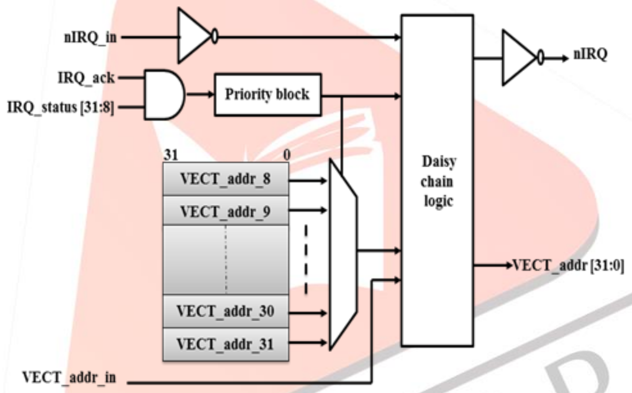


Fig. 6. Block of IRQ Interrupt Logic.

The IRQ interrupt logic generates the vectored IRQ interrupt signal by combining the IRQ interrupt requests in the interrupt controller and any requests from daisy-chained interrupt controllers. This signal is then sent to the IRQ vector address and priority logic. Fig.6. shows a block diagram of the IRQ interrupt logic.
Here we use fixed priority logic for lower 24 bits of INTSOURCE and generates the nIRQ signals which is active low and selects the VECT_addr of the respective peripheral from vectored table to the CPU.
IRQ_status acts as a select line for vectored address selection. nIRQ is active low signal for CPU.

3) Interrupt Priority Block

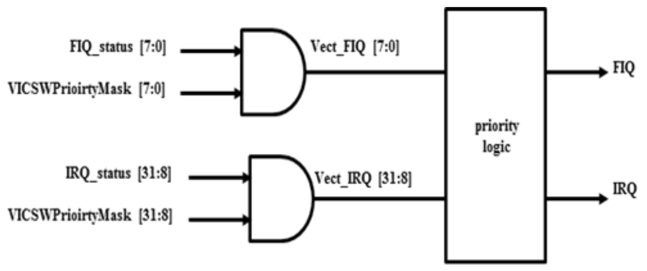


Fig. 7. Block of Interrupt Priority.

The interrupt priority block prioritizes the interrupt requests using the programmed and hardware priority levels in the following order:

- Vectored interrupt requests.
- External interrupt requests (from daisy chain).

The priority of each of the vectored interrupts is programmable, allowing the order in which interrupts are serviced to be dynamically changed. This is done by programming the values in the Vector Priority Registers.

Daisy Chain Interface

A daisy chain is an interconnection of computer devices, peripherals, or network nodes in series, one after another. It is the computer equivalent of a series electrical circuit. The main advantage of the daisy chain is its simplicity. Another advantage is scalability. The user can add more nodes anywhere along the chain, up to a certain maximum.

To increase the number of input bits of the Interrupt Controller we need to cascade the interrupt controller, for this we use the Daisy chain concept. In this the Interrupt Controller is Daisy chained.

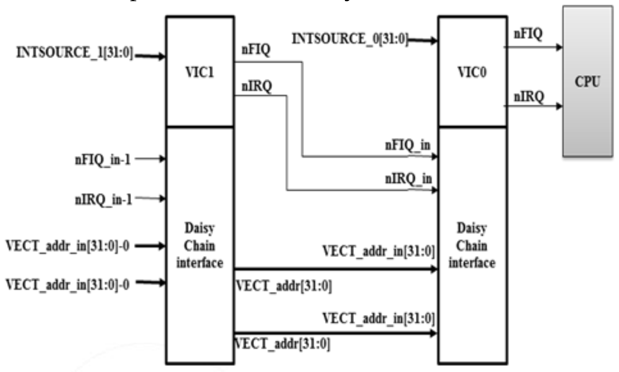


Fig. 8. Block of Daisy chain interface.

Fig.8. shows the connections between the two VICs and the processor when we use them in a daisy-chain. This configuration is referred to as VIC0 blocking mode.

In this mode, VIC0 is totally responsible for blocking lower level interrupts from the daisy chained VICs [3].

AHB Slave Interface

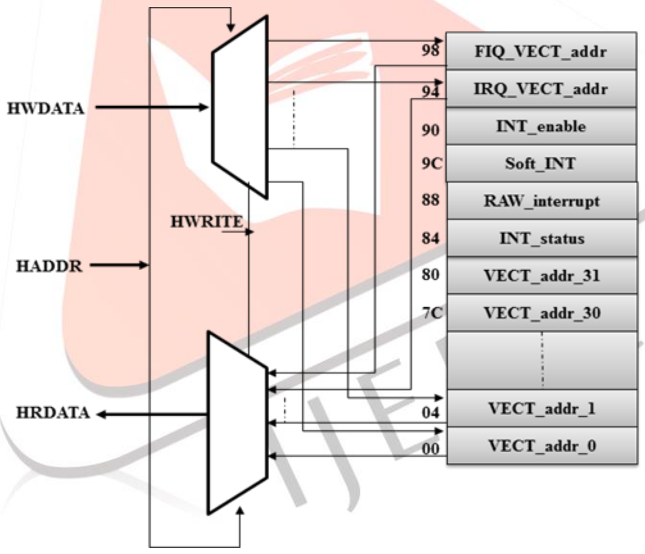


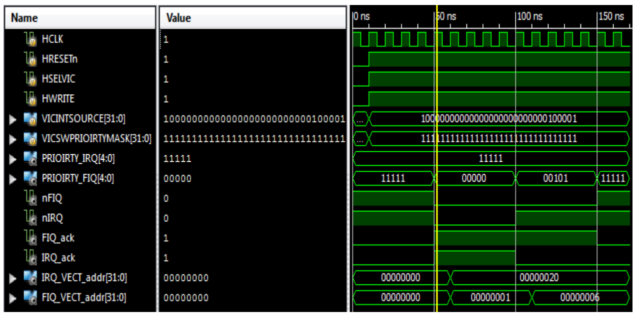
Fig. 9. Block of AHB slave interface.

The AMBA AHB bus protocol is designed to be used with a central multiplexer interconnection scheme. Master device drive out the address and control signals indicating the transfer they wish to perform. The slave device passively waits for address and control signal from the master device. When the slave device is selected, it has to provide the response signals indicating the transfer status [2].

Fig.9. shows the block of AHB slave interface. It maps the memory configuration space with the Interrupt Controller and performs the data transaction as AHB asserts it signals.

V. Simulation Results

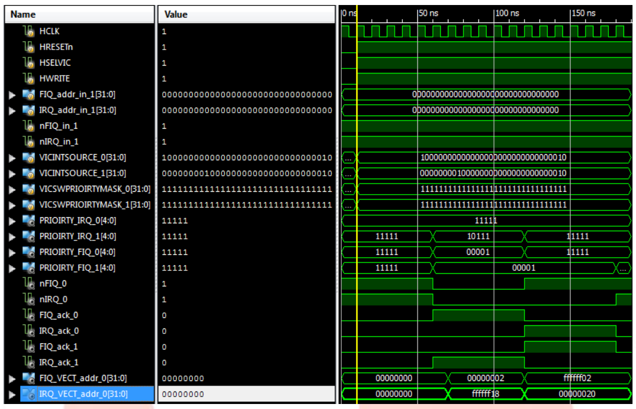
Fig.10. shows the simulation result of Interrupt Controller which generates FIQ and IRQ interrupt request signals.



Name	Value
HCLK	1
HRESETn	1
HSELVIC	1
HWRITE	1
VICINTSOURCE[31:0]	10000000000000000000000000100001
VICSWPRIOIRTYMASK[31:0]	11111111111111111111111111111111
PRIOIRTY_IRQ[4:0]	11111
PRIOIRTY_FIQ[4:0]	00000
nFIQ	0
nIRQ	0
FIQ_ack	1
IRQ_ack	1
IRQ_VECT_addr[31:0]	00000000
FIQ_VECT_addr[31:0]	00000000

Fig. 10. Interrupt Controller.

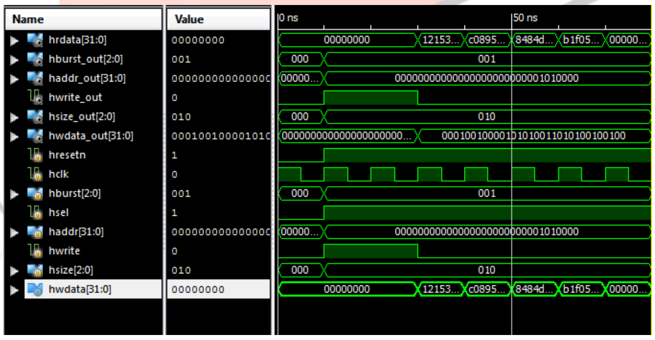
Fig.11. shows the simulation result of Daisy Chained Interrupt Controller which generates FIQ and IRQ interrupt request signals.



Name	Value
HCLK	1
HRESETn	1
HSELVIC	1
HWRITE	1
FIQ_addr_in_1[31:0]	00000000000000000000000000000000
IRQ_addr_in_1[31:0]	00000000000000000000000000000000
nFIQ_in_1	1
nIRQ_in_1	1
VICINTSOURCE_0[31:0]	10000000000000000000000000000010
VICINTSOURCE_1[31:0]	00000000100000000000000000000010
VICSWPRIOIRTYMASK_0[31:0]	11111111111111111111111111111111
VICSWPRIOIRTYMASK_1[31:0]	11111111111111111111111111111111
PRIOIRTY_IRQ_0[4:0]	11111
PRIOIRTY_IRQ_1[4:0]	11111
PRIOIRTY_FIQ_0[4:0]	11111
PRIOIRTY_FIQ_1[4:0]	11111
nFIQ_0	1
nIRQ_0	1
FIQ_ack_0	0
IRQ_ack_0	0
FIQ_ack_1	0
IRQ_ack_1	0
FIQ_VECT_addr_0[31:0]	00000000
IRQ_VECT_addr_0[31:0]	00000000

Fig. 11. Daisy chained Interrupt Controller.

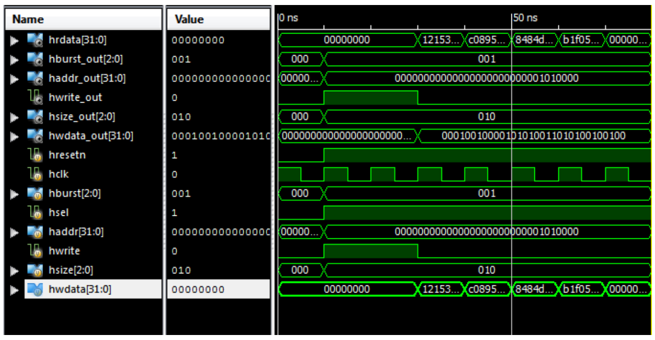
Fg.12. shows the simulation result of AHB interface write operation.



Name	Value
hrdata[31:0]	00000000
hburst_out[2:0]	001
haddr_out[31:0]	000000000000000
hwrite_out	0
hsize_out[2:0]	010
hwdata_out[31:0]	000100100001010
hresetn	1
hclk	0
hburst[2:0]	001
hsel	1
haddr[31:0]	000000000000000
hwrite	0
hsize[2:0]	010
hwdata[31:0]	00000000

Fig. 12. AHB write cycle.

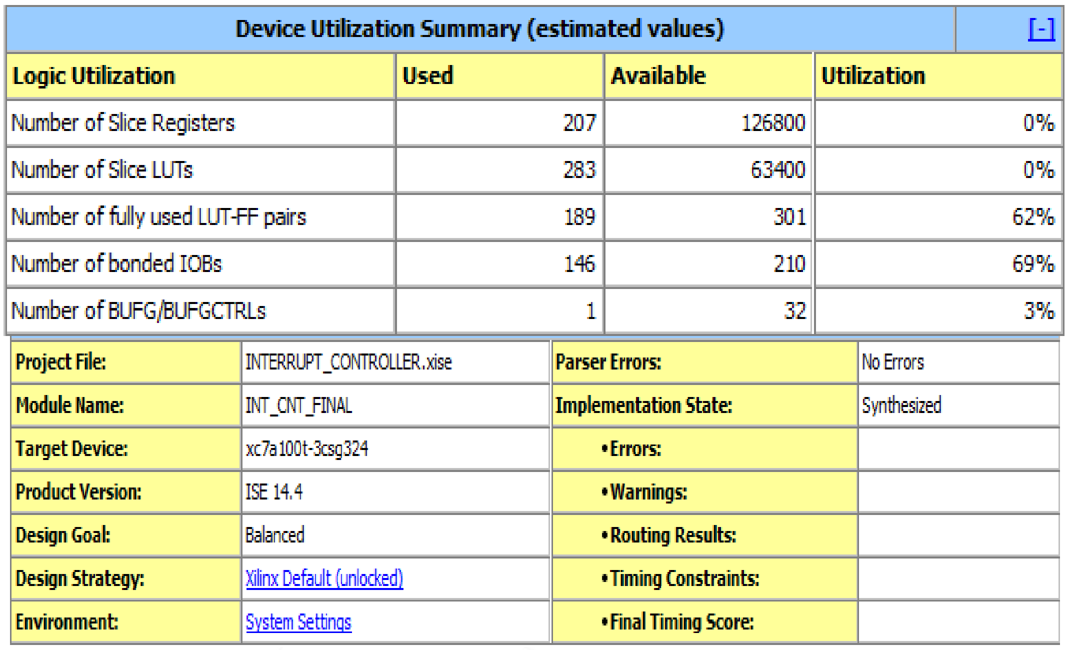
Fg.13. shows the simulation result of AHB interface read operation.



Name	Value
hrdata[31:0]	00000000
hburst_out[2:0]	001
haddr_out[31:0]	000000000000000
hwrite_out	0
hsize_out[2:0]	010
hwdata_out[31:0]	000100100001010
hresetn	1
hclk	0
hburst[2:0]	001
hsel	1
haddr[31:0]	000000000000000
hwrite	0
hsize[2:0]	010
hwdata[31:0]	00000000

Fig. 13. AHB write cycle.

VI. Implementation Results



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	207	126800	0%
Number of Slice LUTs	283	63400	0%
Number of fully used LUT-FF pairs	189	301	62%
Number of bonded IOBs	146	210	69%
Number of BUFG/BUFGCTRLs	1	32	3%

Project File:	INTERRUPT_CONTROLLER.xise	**Parser Errors:**	No Errors
Module Name:	INT_CNT_FINAL	**Implementation State:**	Synthesized
Target Device:	xc7a100t-3csg324	**•Errors:**	
Product Version:	ISE 14.4	**•Warnings:**	
Design Goal:	Balanced	**•Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**•Timing Constraints:**	
Environment:	System Settings	**•Final Timing Score:**	

Fig. 14. Interrupt Controller synthesis result.

Synthesis result of the Integrated Interrupt Controller implementation is shown in Fig.14 with respect to small functions like gates, inverters large units like multiplexers, encoders [9]. The synthesis is done using the Xilinx ISE simulator.

VII. RTL Schematic View

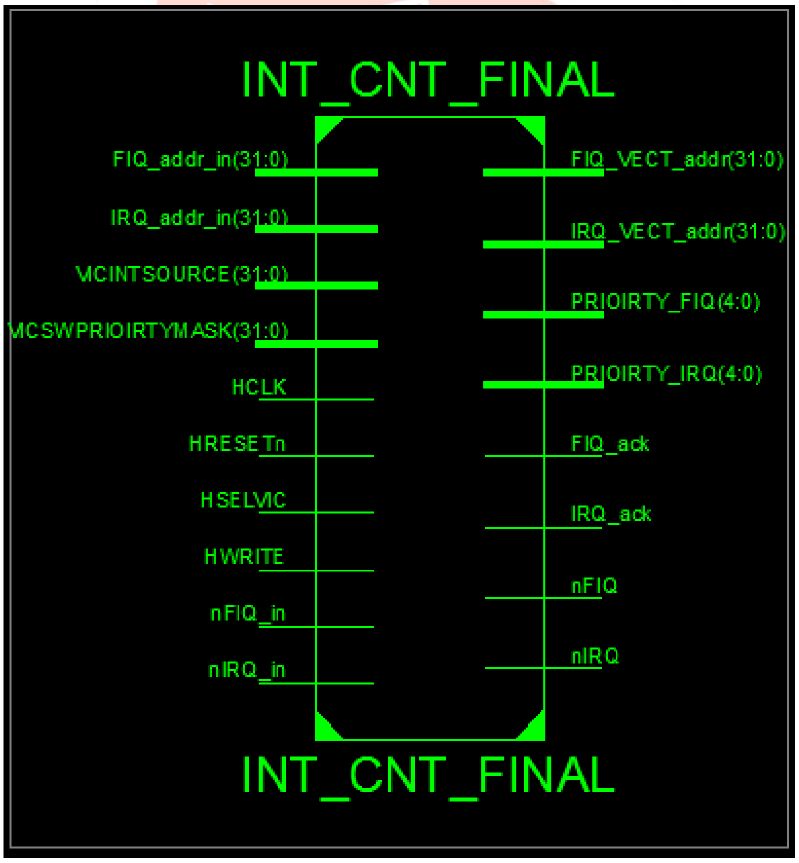


Fig. 15. Interrupt Controller synthesis result.

RTL schematic view of the synthesized design is shown in the Fig 15.

VIII. CONCLUSIONS

The proposed work involves the design of Interrupt Controller. Interrupt-based programming is widely used for interfacing a processor with peripherals and allowing software threads to interact. This work will present the use of Interrupt Controller which adopts Vector Interrupts and Daisy chain. This design efficiently distributes multiple interrupts on a processor. In addition, it supports several features useful in system. This IP sits onto the AMBA bus, which is one of the most used buses in any advanced System On Chip (SoC). The prioritized interrupts, also supported by the Interrupt Controller, reduce the latency of high priority interrupts, and that might also be considered "more efficient".

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression, "One of us (R. B. G.) thanks . . ." Instead, try "R. B. G. thanks". Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

REFERENCES

[1] ARM Corporation, AMBA specification 3.0, reference manual, 2006.

[2] Wei Chipin,, "Design of a Configurable Multichannel Interrupt Controller" — 2010 Second Pacific-Asia Conference on Circuits, Communications and System (PACCS)

[3] ARM Corporation, Prime cell Vectored interrupt controller PL192, reference manual, 2002.

[4] Vectored Interrupt Controller Usage and Applications-www.altera.com/literature/an/AN595.pdf.

[5] About Interrupt controller- www.arm.com/products/system-ip/controllers/interrupt.php.

[6] Vectored Interrupt Controller Implementation of Advanced Bus Architecture on FPGA, Vasanth H, Dr.A.R.Aswath

[7] About Advanced Microcontroller Bus Architecture-en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture.

[8] Digital Design principles and practices, John F. Wakerly, third edition, Prentice Hall Publication, 2000.

[9] J. Basker, A Verilog HDL Synthesis 1st Edition, 1998.